Concurrent Programming

C#
Parallel Class

Parallel Class

The Parallel class in C# is part of the System.Threading.Tasks namespace and is used for executing parallel operations.

It enables easy launching of multiple tasks simultaneously, which can significantly speed up program execution - especially for coarse-grained operations that are independent of each other.

Parallel Class

Key Methods in the Parallel Class (C#):

- **Parallel.For** – Executes a for loop in parallel. Each iteration can run on a separate thread.
- **Parallel.ForAsync** – Asynchronous version of Parallel.For, returns a task representing the parallel execution.
- **Parallel.ForEach** – Processes elements of a collection in parallel. Each item can be handled on a separate thread.
- **Parallel.ForEachAsync** – Asynchronous counterpart to Parallel.ForEach, suitable for non-blocking operations.
- **Parallel.Invoke** – Executes multiple actions in parallel. Each action can run on a separate thread.

Parallel Class

The simplest example

- Classic loop

```
for (int i = 0; i < max; i++)
{
     DoSomething(i);
}
```

- *Parallel version*

```
Parallel.For(0, max, (i) => DoSomething(i));
```

- Simplified *Parallel version*

```
Parallel.For(0, max, DoSomething);
```

Parallel Class

The example

```
ThreadPool.SetMinThreads(64, 64);
Console.CursorVisible = false;
int max = Console.WindowHeight;
//anonymous version
Parallel.For(0, max, iter =>
 {
    for (int i = 0; i < Console.WindowWidth; i++)
    {
    lock (Console.Out)
        {
        Console.ForegroundColor = (ConsoleColor)(iter % 16);
        Console.SetCursorPosition(i, iter % Console.WindowHeight)
        Console.Write("#");
        }
    Thread.Sleep(Random.Shared.Next(50));
    }
});
```


Example: LecureParallel/SimpleHashPrinting2

Parallel Class

Breaking

- In a regular loop, we use `break` to exit the loop.
- In the Parallel version, stopping all running iterations requires a coordinated mechanism.
- We can pass a `ParallelLoopState` object to the action delegate. Using this object, we can call:
  - `Break()`
    - Signals that no further iterations beyond the current index should start.
    - Sets the ShouldExitCurrentIteration flag.
    - Allows the current iteration to complete gracefully.
  - `Stop()`
    - Immediately requests that all iterations stop as soon as possible.
    - Also sets ShouldExitCurrentIteration, but more aggressively than Break().

Example: LecureParallel/SimpleHashPrinting2

Parallel Class

Breaking example

```
static Action<int,ParallelLoopState> PrintHashes = (iter,state) =>
{
    for (int i = 0; i < Console.WindowWidth; i++)
    {
        if (i == 50 && iter == 10)
        {
            state.Break();
            //return; //it will work as state.Stop();
        }
        //if (i == 50 && iter == 10) state.Stop();
        if (state.ShouldExitCurrentIteration)
        {
            return;
        }
        lock (Console.Out)
        {
            Console.ForegroundColor = (ConsoleColor)(iter % 16);
            Console.SetCursorPosition(i, iter % Console.WindowHeight);
            Console.Write((i % 10).ToString());
        }
        Thread.Sleep(Random.Shared.Next(200+50* iter)); //with asymetric delay
    }
};
```

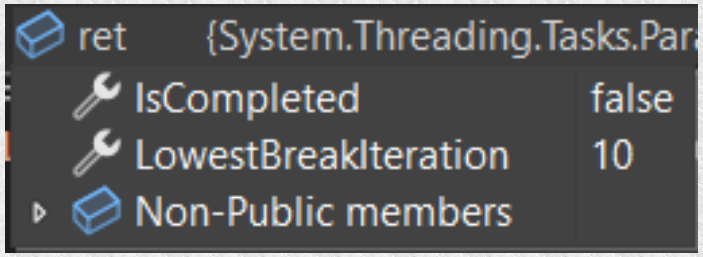
We assume the loop will break at the 10th iteration out of 50.

Also interrupt the current iteration

If a cancellation request has been issued by another task, we can respond to it here.

Parallel Class

- Parallel.For Returns a `ParallelLoopResult` object
- you can inspect the returned ParallelLoopResult object to determine:
  - `IsCompleted` - Indicates whether the loop ran to completion without any Break() or Stop() calls.
  - LowestBreakIteration – Returns the smallest iteration index at which Break() was called. If no break occurred, this value is null.



Example: LecureParallel/SimpleHashPrinting3

Parallel Class

Degree Of Parallelism

- You can pass additional parameters to a Parallel.For loop using the `ParallelOptions` class. One of the key options is:
- `MaxDegreeOfParallelism` – Specifies the maximum number of concurrent tasks that can be used to execute the loop. This allows you to control the level of parallelism and limit resource usage.

```
int iterations = Console.WindowHeight;
ParallelOptions options = new ParallelOptions();
options.MaxDegreeOfParallelism = iterations;

 ParallelLoopResult ret = Parallel.For(1, iterations,
options, (i, state) =>
{ PrintHashes(i, state); });
```


Example: LecureParallel/SimpleHashPrinting3

Parallel Class

Asynchronous Version

- The asynchronous version of the parallel loop is ForAsync.
- It returns a running Task that can be awaited to retrieve the result.
- There are three overloads of ForAsync:

- ```
  public static Task ForAsync<T>(T fromInclusive, T
  toExclusive, Func<T, CancellationToken, ValueTask> body)
   where T : notnull, IBinaryInteger<T>
  ```

- ```
  public static Task ForAsync<T>(T fromInclusive, T
  toExclusive, CancellationToken cancellationToken, Func<T,
  CancellationToken, ValueTask> body)
    where T : notnull, IBinaryInteger<T>
  ```

- ```
  public static Task ForAsync<T>(T fromInclusive, T
  toExclusive, ParallelOptions parallelOptions, Func<T,
  CancellationToken, ValueTask> body)
    where T : notnull, IBinaryInteger<T>
  ```
  ```

Parallel Class

Asynchronous Version

- In the asynchronous version, `Parallel.ForAsync` returns the running task.
- It can run in the background, or we can wait for it to finish at a convenient time.
- We can also pass a `CancellationToken` or `ParallelOptions.`
- All three overloads receive an iteration range from to and a function to which we pass the iteration number and a cancellationToken instance.
- The function also returns a `ValueTask`.

```
static  ValueTask PrintHashes(int iter, CancellationToken ctoken)
```

Parallel Class

ValueTask vs Task

- **Task** is a reference type allocated on the heap. Each instance requires memory allocation, which can lead to higher memory consumption and more frequent garbage collection. It is more versatile and can be used in most asynchronous scenarios. It has built-in exception handling and can store exceptions that occur during the execution of an asynchronous operation.
- **ValueTask** is a value type allocated on the stack. It can be more efficient in terms of memory management because it avoids heap allocations for completed asynchronous operations. It is primarily used in scenarios where asynchronous operations often complete immediately, thus avoiding the cost of memory allocation. It has certain limitations and should not be used in cases where an asynchronous operation may be awaited multiple times, as this can lead to unexpected behavior.

Parallel Class

Task vs ValueTask, when use it

Task

- In most asynchronous scenarios, where operations can take longer.
- When you need full exception handling and a more versatile approach.
- When an asynchronous operation may be awaited multiple times..

ValueTask

- When an asynchronous operation often completes immediately.
- When you want to minimize memory allocations and reduce the load on the garbage collector.
- When you are certain that an asynchronous operation will not be awaited multiple times.

Parallel Class

ValueTask example

```
static  ValueTask PrintHashes(int iter, CancellationToken ctoken)
{
    return new ValueTask(Task.Run(() =>
    {
        for (int i = 0; i < Console.WindowWidth; i++)
        {
            if (ctoken.IsCancellationRequested) break;
            lock (Console.Out)
            {
               Console.ForegroundColor = (ConsoleColor)(iter % 16);
               Console.SetCursorPosition(i, iter %
Console.WindowHeight);
                Console.Write("#");
            }
            Thread.Sleep(Random.Shared.Next(100));
        }
    }));
}
```


Example: LecureParallel/SimpleHashPrinting4

Parallel Class

ValueTask example

```
CancellationTokenSource cancellationSource = new CancellationTokenSource();
int iterations = 32;
ParallelOptions options = new ParallelOptions()
    {
    CancellationToken = cancellationSource.Token ,
    MaxDegreeOfParallelism = iterations
    };
Task ret = Parallel.ForAsync(0, iterations, options, PrintHashes);
cancellationSource.CancelAfter(2000);
try
{
    ret.Wait();
}
catch (Exception ex)
{
    Console.SetCursorPosition(0, iterations);
    Console.ForegroundColor = ConsoleColor.White;
    Console.WriteLine($"We have exception {ex.Message}");
}
```


Example: LecureParallel/SimpleHashPrinting4

Parallel Class

TaskScheduler

```
 sealed class TwoThreadTaskScheduler : TaskScheduler,
IDisposable
   {
   ...
   }
```

```
ParallelOptions options = new ParallelOptions()
  {   //custom taskscheduler
      TaskScheduler = new TwoThreadTaskScheduler(6),
      MaxDegreeOfParallelism = iterations
  };
  Parallel.For(0, iterations,options, PrintHashes);
```


Example: LecureParallel/SimpleHashPrinting7

Parallel Class

- *Invoke* executes multiple actions simultaneously.

```
static void PrintAsterisk()
{
    for (int i = 0; i < Console.WindowWidth; i++)
    {
        lock (Console.Out)
        {
            Console.ForegroundColor = (ConsoleColor)(Random.Shared.Next(1, 16));
            Console.SetCursorPosition(Random.Shared.Next(0, Console.WindowWidth),
Random.Shared.Next(0, Console.WindowHeight));
            Console.Write("*");
        }
        Thread.Sleep(Random.Shared.Next(50));
    }
}
static void Main(string[] args)
{
    ThreadPool.SetMinThreads(64, 64);
    Console.CursorVisible = false;
    Parallel.Invoke(PrintAsterisk, PrintAsterisk, PrintAsterisk, PrintAsterisk);
    Console.SetCursorPosition(0, Console.WindowHeight);
    Console.ForegroundColor = ConsoleColor.White;
}
```

Concurrent execution of the provided actions

Example: LecureParallel/SimpleAsteriskPrinting1

Parallel Class

Parallel.Invoke

- Wywołanie całej tablicy akcji

```
static Action<int> PrintHashes = (iter) =>
{
  ...
}

int threadsNo = Console.WindowHeight - 1; ;

Action[] actions = new Action[threadsNo];
for (int i = 0; i < threadsNo; i++)
{
    int iter = i;
    actions[i] = () => PrintHashes(iter);
}
Parallel.Invoke(actions);
```

Concurrent execution of the entire set of actions

Example: LecureParallel/SimpleHashPrinting5

Parallel Class

Parallel.Invoke asynchroniczne

```
  async Task InvokeAsync(params Action[] actions)
{
    var tasks = actions.Select(a => Task.Run(a));
    await Task.WhenAll(tasks);
}

static void Main(string[] args)
{
 ThreadPool.SetMinThreads(64, 64);
 Console.CursorVisible = false;
 int threadsNo = Console.WindowHeight;
 Action[] actions = new Action[threadsNo];
 for (int i = 0; i < threadsNo; i++)
    {
        int iter = i;
        actions[i] = () => PrintHashes(iter);
    }
Task runningActions = InvokeAsync(actions);
runningActions.Wait(); //Prevent from ending too early
}
```

Create a task for each action, then return a task that awaits all of them.

Prepare a set of actions

Asynchronous invocation of the entire set of actions

Example: LecureParallel/SimpleHashPrinting6

Parallel For and another

```
int[] numbers = GenerateRandomArray(sizeOfProblem);
int sum = 0;

foreach (int number in numbers) sum += number;
Parallel.ForEach(numbers, number => sum += number);
Parallel.ForEach(numbers, number => { lock (sw) { sum += number; } });
sum = numbers.Sum(number => number);
numbers.AsParallel().ForAll(number => sum += number);
numbers.AsParallel().ForAll(number => { lock (sw) { sum += number; } });

sum = numbers.AsParallel().Sum(number => number);
sum = numbers.AsParallel().Sum();
```


Example: LecureParallel/VerySimpleSum

Parallel For and another

```
 Parallel.For<double>(0, numbers.Length, new ParallelOptions
{ MaxDegreeOfParallelism = 8 },
    // Initialize the local states
    () => 0,
    // Accumulate the thread-local computations in the loop body
    (i, loop, localState) =>
    {
        return localState + numbers[i];
    },
    // Combine all local states
    localState => { lock (numbers) { sum += localState; } }
 );
```


Example: LecureParallel/VerySimpleSum

Parallel For and another

```
var tasks = new List<Task<double>>();
int processorCount = Environment.ProcessorCount;
int end = numbers.Length - 1;
int range = (end + 1) / processorCount;
//   Console.WriteLine($"I found {processorCount} processors");
for (int i = 0; i < processorCount; i++)
{
    int rangeStart = i * range;
    int rangeEnd = (i + 1) * range;
    tasks.Add(Task.Run(() =>
    {
        double localSum = 0;
        for (int j = rangeStart; j < rangeEnd; j++)
        {
            localSum += numbers[j];
        }
        return localSum;

    }));
}
double[] partialResults = Task.WhenAll(tasks).Result;
sum = partialResults.Sum();
```


Example: LecureParallel/VerySimpleSum

Parallel For and another

```
Simple sum sumed by foreach in:                                  271    with result    450001866
Simple sum sumed by unsafe Parallel.Foreach in:                  1038   with result    60346552
Simple sum sumed by Parallel.Foreach with lock in:               4814   with result    450001866
Simple sum sumed by one thread Linq in:                          529    with result    450001866
Simple sum by Linq as Parallel ForAll without lock in:           1217   with result    63061203
Simple sum by Linq as Parallel ForAll  with lock in:             4548   with result    450001866
Simple sum by Linq as Parallel without selector in:              74     with result    450001866
Simple sum  by Parallel.For and LocalState without lock in:      190    with result    176834245
Simple sum by Parallel.For and LocalState with lock in:          173    with result    450001866
Simple sum by 8 tasks in:                                        93     with result    450001866
```


Example: LecureParallel/VerySimpleSum

Parallel For and another

```
sum = numbers.AsParallel().Sum(number => number);
sum = numbers.AsParallel().Sum();
```

```
Simple sum sumed by foreach in:                          265     with result     450012306
Simple sum sumed by unsafe Parallel.Foreach in:          1044    with result     62565649
Simple sum sumed by Parallel.Foreach with lock in:       5996    with result     450012306
Simple sum sumed by one thread Linq in:                  578     with result     450012306
Simple sum by Linq as Parallel ForAll without lock in:   1582    with result     61263259
Simple sum by Linq as Parallel ForAll  with lock in:     6608    with result     450012306
Simple sum by Linq as Parallel with selector in:         431     with result     450012306
Simple sum by Linq as Parallel without selector in:      337     with result     450012306
Simple sum  by Parallel.For and LocalState without lock in:  233  with result    265437820
Simple sum by Parallel.For and LocalState with lock in:  216     with result     450012306
Simple sum by 8 tasks in:                                101     with result     450012306
```


Example: LecureParallel/VerySimpleSum

Parallel For and another

```
static double calculatedValue(double value) //always returns 1
    {
    return (Math.Sin(Math.Sin(value * value) * Math.Cos(value * value)
     / Math.PI) * Math.Sin(Math.Sin(value * value) * Math.Cos(value * value)
     / Math.PI)) > 1 ? 0 : 1;
    }

foreach (int number in numbers)
     sum += calculatedValue(number);
Parallel.ForEach(numbers, number => sum += calculatedValue(number));
Parallel.ForEach(numbers, number => { lock (sw)
                                       { sum += calculatedValue(number); }
                                    });
Parallel.For(0, numbers.Length, i => sum += calculatedValue(numbers[i]));
Parallel.For(0,numbers.Length, i => { lock (sw) { sum +=
calculatedValue(numbers[i]); } });
sum = numbers.Sum(number => calculatedValue(number));
sum = numbers.AsParallel().Sum(number => calculatedValue(number));
```


Example: LecureParallel/SumWithCalculation

Parallel For and another

```
 Parallel.For<double>(0, numbers.Length, //new ParallelOptions
{ MaxDegreeOfParallelism = 8 },
    // Initialize the local states
    () => 0,
    // Accumulate the thread-local computations in the loop body
    (i, loop, localState) =>
    {
        return localState + calculatedValue(numbers[i]);
    },
    // Combine all local states
    localState => { lock (numbers) { sum += localState; } }
 );
```


Example: LecureParallel/SumWithCalculation

Parallel For and another

```
var tasks = new List<Task<double>>();
int processorCount = Environment.ProcessorCount;
int end = numbers.Length - 1;
int range = (end + 1) / processorCount;
for (int i = 0; i < processorCount; i++)
{
    int rangeStart = i * range;
    int rangeEnd = (i + 1) * range;
    tasks.Add(Task.Run(() =>
    {
        double localSum = 0;
        for (int j = rangeStart; j < rangeEnd; j++)
        {
            localSum += calculatedValue(numbers[j]);
        }
        return localSum;

    }));
}
double[] partialResults = Task.WhenAll(tasks).Result;
sum = partialResults.Sum();
```


Example: LecureParallel/SumWithCalculation

Parallel For and another

```
One thread foreach in:                              755     with result 10000000
Unsafe Parallel.Foreach in:                         224     with result 1247965
Parallel.Foreach with lock in:                      1229    with result 10000000
Unsafe Parallel.For in:                             215     with result 1180724
Parallel.For with lock in:                          1217    with result 10000000
Simple one thread Linq in:                          779     with result 10000000
Linq as Parallel in:                                220     with result 10000000
Parallel.For and LocalState without lock in:        184     with result 8709343
Parallel.For and LocalState with lock in:           176     with result 10000000
Simple 8 tasks in:                                  171     with result 10000000
```


Example: LecureParallel/SumWithCalculation